Chapter 18

Release Planning (Longer-Term Planning)

Release planning is longer-term planning that enables us to answer questions like "When will we be done?" or "Which features can I get by the end of the year?" or "How much will this cost?" Release planning must balance customer value and overall quality against the constraints of scope, schedule, and budget. In this chapter I discuss how release planning fits into the Scrum framework and how to perform release planning on both fixed-date and fixed-scope releases.

Overview

Every organization must determine the proper cadence for releasing features to its customers (see Figure 18.1).

Though the output of a sprint is potentially shippable, many organizations choose not to release new features after every sprint. Instead, they combine the results of multiple sprints into one release.

Other organizations match the release cadence to the sprint cadence. In such cases the organization releases the potentially shippable product increment created during that sprint at the end of each sprint.

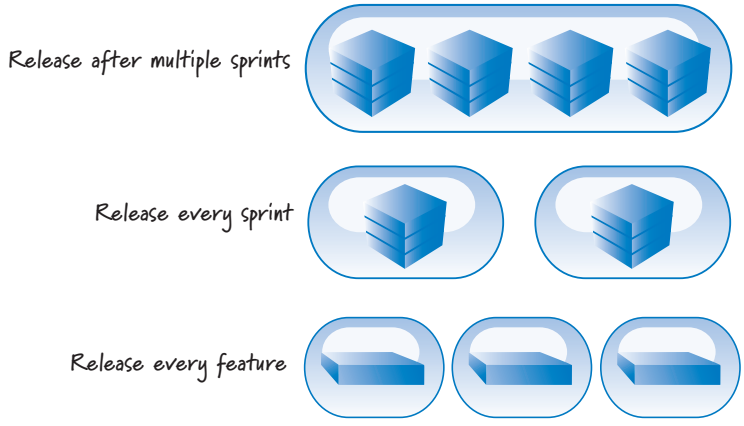

FIGURE 18.1 Different release cadences

Some organizations don't even wait for the sprint to end; they release each feature as it is completed, a practice often referred to as **continuous deployment** (or **continuous delivery**). Such organizations release a feature or a change to a feature to some or all of their customers as soon as the feature is available, which might be as often as several times a day.

Whether the organization intends to deploy every sprint, every few sprints, or continuously, most organizations find some amount of longer-term, higher-level planning to be useful. I refer to this type of planning as release planning. If the term *release planning* seems inappropriate to your context, replace the term with one that is better suited. Synonyms I have heard different organizations use include

- Longer-term planning—connoting that the goal is to look at a horizon that is greater than a single sprint
- Milestone-driven planning—because releases tend to align with significant milestones, such as an important user conference or the completion of a minimum set of features for a viable, marketable release

Whatever you choose to call it, release planning targets a future state when important variables such as date, scope, and budget need to be balanced.

Timing

Release planning is not a one-time event but rather a frequent, every-sprint activity (see Figure 18.2). Initial release planning logically follows envisioning/product-level planning.

The purpose of product planning is to envision what the product should be; the aim of release planning is to determine the next logical step toward achieving the product goal.

Before starting a release, many organizations that use Scrum conduct initial release planning to create a preliminary release plan. Normally this activity lasts a day or two, but the duration varies based on the size and risk of the release and the participants' familiarity with what is being created.

When developing a new product, this initial release plan won't be complete or too precise. Instead, as validated learning becomes available during the release, we update the release plan. Release plans can be revised as part of each sprint review or in the normal course of preparing for and conducting each subsequent sprint.

Participants

Release planning involves collaboration between the stakeholders and the full Scrum team. At some point all of these people have to be involved because they'll need to make business and technical trade-offs to achieve a good balance of value and quality. Each person's exact involvement over time may vary.

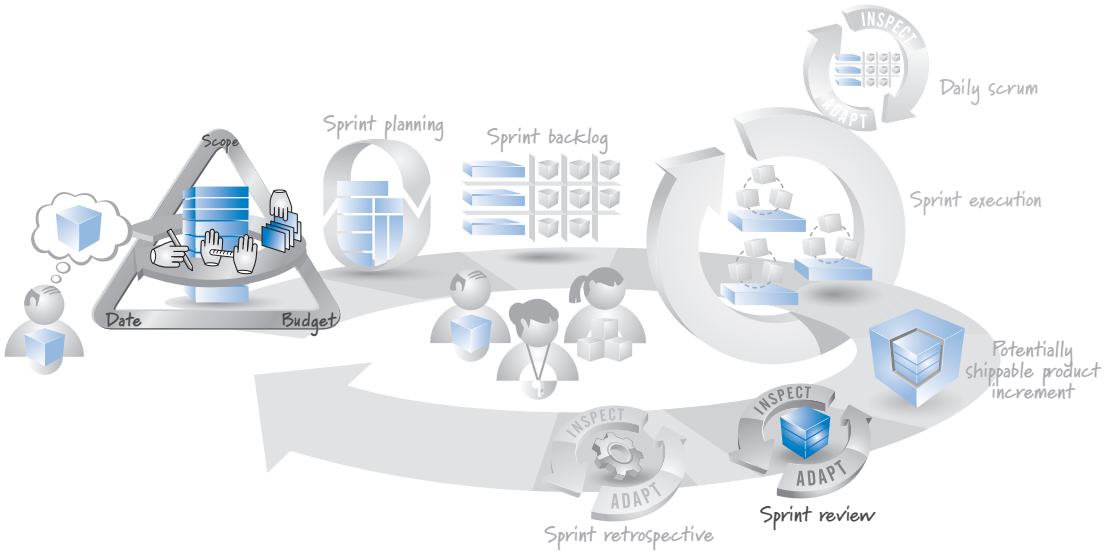

FIGURE 18.2 When release planning happens

Process

Figure 18.3 illustrates the release-planning activity.

The inputs to release planning include outputs from product planning, such as the product vision, high-level product backlog, and product roadmap. We also need the velocity of the team or teams that will work on the release. For an existing team, we use the team's known velocity; otherwise, we forecast the team's velocity during release planning (as described in Chapter 7).

One activity that recurs in release planning is to confirm the release constraints of scope, date, and budget and to review them to see if any changes should be made, given the passage of time and what we now know about the product and this release.

Another activity of release planning is product backlog grooming, which includes creating, estimating, and prioritizing more detailed product backlog items from high-level product backlog items. These activities could occur at multiple points in time:

- After product planning but before initial release planning
- As part of the initial release-planning activity
- During each sprint as necessary (see Chapter 6 for more details on product backlog grooming)

Each release should have a well-defined set of minimum releasable features (MRFs). The initial MRFs for a release might have been defined during envisioning.

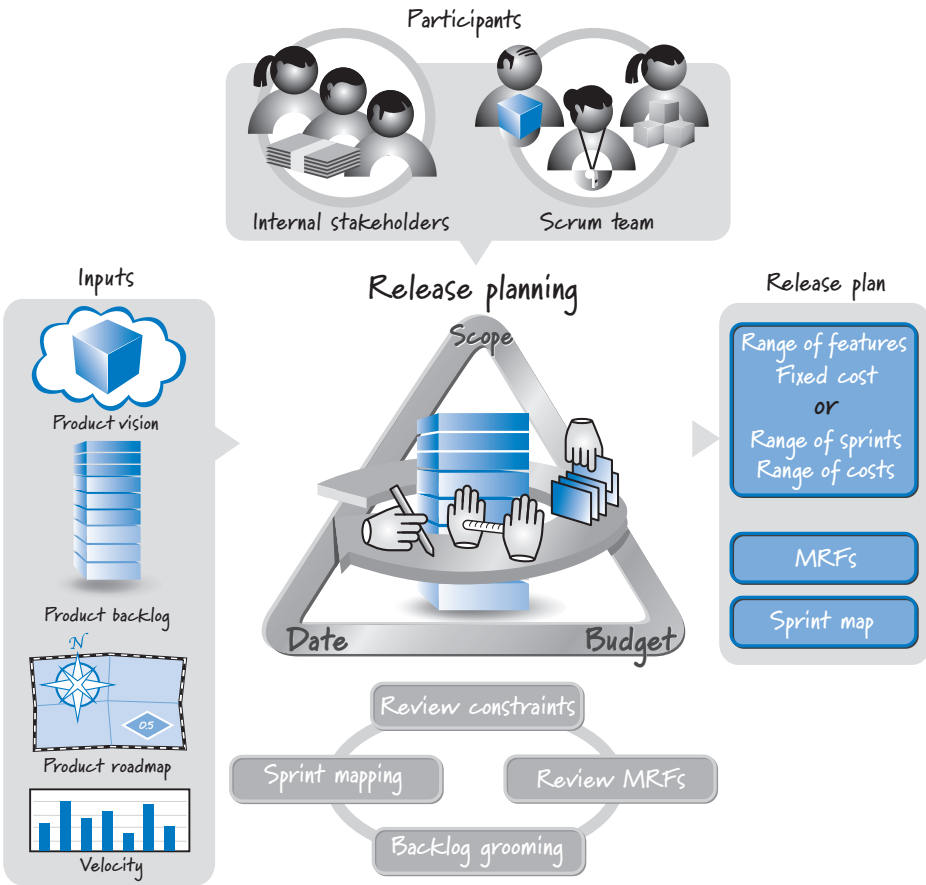

FIGURE 18.3 Release-planning activity

Even so, during release planning we always review the MRFs to ensure that they truly do represent the minimum viable product from the customers' perspective.

During release planning many organizations also produce a sprint map, indicating in which sprint some or many of the product backlog items might be created. A sprint map isn't intended to project too far into the future. Instead, a sprint map is useful for visualizing the near-term future to help us better manage our team's own dependencies and resource constraints, as well as coordinate the efforts of multiple collaborating teams.

The outputs of release planning are collectively referred to as the release plan. The release plan communicates, to the level of accuracy that is reasonable given where we are in the development effort, when we will finish, what features we will get, and what

the cost will be. This plan also communicates a clear understanding of the desired MRFs for the release. Finally, it frequently will show how some of the product backlog items map to sprints within the release.

Release Constraints

The goal of release planning is to determine what constitutes the most valuable next release and what the desired level of quality is. The constraints of scope, date, and budget are important variables that affect how we will achieve our goal.

Based on product planning, one or more of these constraints will probably be established. Chapter 17 introduced the fictional company Review Everything, Inc. In that chapter, we followed Review Everything through envisioning a new product, SR4U, a trainable search agent for online reviews. In the product roadmap for SR4U, Roger and his team determined that it would be advantageous to release the first version of SR4U at an upcoming conference, the Social Media Expo. Thus, SR4U Release 1.0 has a fixed-date constraint—the release must be ready by a certain date: the Social Media Expo. The other constraints (scope and budget) are flexible.

Table 18.1 illustrates different combinations where these constraints are either fixed or flexible.

Let's review the different combinations in light of how they can affect release planning.

Fixed Everything

As I described in Chapter 3, traditional, plan-driven, predictive development approaches assume that the requirements are known or can be predicted up front and that the scope won't change. Based on these beliefs, we can create a complete plan and then estimate the cost and schedule. In Table 18.1 this approach is called "fixed everything."

In Scrum, we don't believe it's possible to get it all right up front; consequently, we also contend that a fixed-everything approach probably won't work. When doing

TABLE 18.1 Development Constraint Combinations

Project Type	Scope	Date	Budget
Fixed everything (not recommended)	Fixed	Fixed	Fixed
Fixed scope and date (not recommended)	Fixed	Fixed	Flexible
Fixed scope	Fixed	Flexible	Fixed (not really)
Fixed date	Flexible	Fixed	Fixed

release planning for product development with Scrum, we require at least one of these variables to be flexible.

Fixed Scope and Date

One approach is to fix both the scope and the date and let the budget be flexible. This approach suffers from a number of issues. First, in many organizations, increasing a budget once development has begun isn't very easy or likely. Second, in my experience, this approach locks down two variables that are very difficult to predefine. And, in practice, even if we start off believing we have a fixed-scope-and-date release, one of them will give.

Take the Y2K issue as an example. Many organizations working on mitigating the Y2K issues had a fixed set of applications that needed to be updated no later than December 31, 1999. Many fixed time and scope; the budget was their variable. In the end, however, they knew that no matter how much they increased their budget, they still weren't going to complete all the work by the hard deadline of December 31. The date wasn't moving, so the scope did. In a sense, the variables of date and scope are constantly playing a game of chicken with each other! (See Figure 18.4.)

At some point when time starts running out, either the scope or the date needs to give way. If neither does, the resulting crash will likely generate large technical debt.

Fixing scope and date and allowing the budget to be flexible assumes that applying more resources to a problem will increase the amount of work we accomplish and/or reduce the amount of time it takes to perform the work. There are certainly instances during product development when this is true. For example, we might choose to spend extra money to expedite when a piece of work is done (perhaps pay a subcontractor more money to do our work before someone else's work). In this case we spend money to buy time.

However, buying time or scope will go only so far. Frequently the work in product development is incompressible—meaning adding more resources or spending more money won't help and might even hurt. This is exemplified cleverly by Fred Brooks: "Nine women don't make a baby in a month" (Brooks 1995).

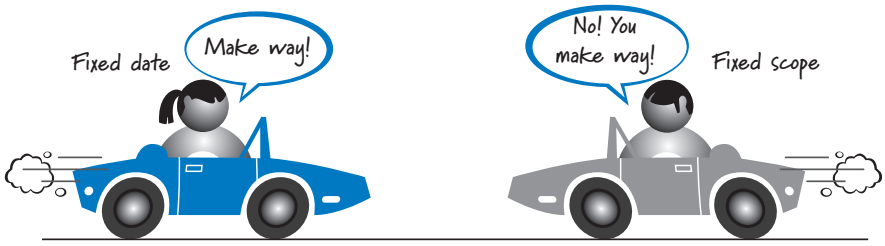

FIGURE 18.4 Fixed date and fixed scope playing a game of chicken

During product development, "flexible budget" frequently translates into "add more people." However, as Brooks warns and as many of us have experienced, "adding manpower to a late software project makes it later" (Brooks 1995). There are times when adding people with the proper skills early in the release might help. Throwing bodies at the problem late, however, will rarely help a fixed-scope-and-date release succeed.

The reality in many organizations is that a flexible budget rarely means adding more people. Usually it means the same people working more hours, especially if these people are salaried employees. Extensive overtime to meet fixed-scope and fixed-date constraints will burn out our staff and violate the Scrum principle of sustainable pace.

If we do find ourselves working on a release that is initially defined as fixed scope and fixed date, Scrum's iterative and incremental approach allows us to understand sooner when we're in trouble, providing more time to rebalance the constraints of scope, date, and budget to achieve a successful outcome.

So far, I've discussed the idea that fixed-everything and fixed-scope-and-date releases are overconstraining for product development. That leaves us two other realistic options: a fixed-scope or a fixed-date release.

Fixed Scope

A fixed-scope model is appropriate where the scope truly is more important than the date. In this model, when we run out of time and we haven't completed all of the features, we extend the date to ensure that we get everything required to meet the MRFs criteria. I don't refer to this model as fixed scope and fixed budget because the budget can't really be fixed; if we give the team more time to finish, people expect to be paid! In other words, if we provide more time to complete the fixed scope, we also have to provide more budget to pay people during that extra time.

Frequently, a fixed-scope scenario exists because the overall scope is too large. A better solution might be to consider smaller, more frequent fixed-date releases. Also, in organizations where multiple groups (such as development, marketing, and support) must coordinate activities, moving the date can be very disruptive to the other groups' plans. Even so, I discuss later in this chapter how to plan a fixed-scope release using Scrum in case you find yourself in a situation where the scope is more important than the date.

Fixed Date

Fixed date is the final approach shown in Table 18.1. Many people, myself included, consider this to be the approach most closely aligned with Scrum principles. Simply put, we can fix both the date and the budget, but the scope must be flexible.

The Scrum principle of creating the highest-priority features first should lessen any perceived pain of having to drop features. When we run out of time on a

fixed-date release, whatever hasn't yet been built should be of lower value than what has already been built. It is much easier to make a decision to ship if the features that are missing are low value. If we are missing high-value features, we'll most likely extend the date if we can.

This works only when the high-priority features are truly done, per our agreed-upon definition of done. We don't want a scenario where the high-value, must-have features are really only 75% to 90% done and then we then have to drop one or two of them from the release in order to get the others to the 100% done level.

A fixed-date model becomes even easier to use if we can define a set of minimum releasable features that truly is small. If we can comfortably deliver the MRFs by the fixed date, we are in good shape, because any other features, by definition, are only nice-to-have features.

Fixed-date releases also dovetail nicely with the Scrum emphasis on timeboxing. By establishing a fixed amount of time for the release, we constrain the amount of work we can do and force people to make the difficult prioritization decisions that have to be made.

Variable Quality

If we overly constrain scope, date, and budget, quality becomes "flexible." This can lead us to deliver a solution that fails to meet customer expectations. Or, as I discussed in Chapter 8, flexible quality can result in the accrual of technical debt, which makes it more difficult in the future to add to or adapt our product.

Updating Constraints

An important part of ongoing release planning is to take our current knowledge and revisit these constraints to see if they should be rebalanced. For example, what should Roger and his team at Review Everything do if they approach the SR4U Release 1.0 deadline and it's clear that they won't complete the minimum releasable features? Because this is a fixed-date release, a good first strategy is to drop lower-value features. Let's assume, though, that in this case they would have to drop must-have features that are part of the MRFs in order to meet the date constraint.

Perhaps the right solution is to define a smaller set of features that are included in the MRFs. For example, the initial version of SR4U might focus on filtering restaurant reviews from only a small number of fixed sources. Roger and his team need to assess whether narrowing the scope degrades perceived customer value to an unacceptable level. And, if it is decided that Review Everything can't drop features without substantially damaging value, the company might consider adding more people (change the budget) or giving up on the hope of launching the service at the Social Media Expo (change the date).

These are the decisions that we must continuously make, revisit, and then make again during any development effort.

Grooming the Product Backlog

A fundamental activity of release planning is grooming the product backlog to meet our value and quality objectives. During envisioning (product planning) we create a high-level product backlog (perhaps with epic-level stories) and then use it to define a set of minimum releasable features for each release. Many of these backlog items are too large to be useful during release planning.

For example, during SR4U envisioning Roger provided a rough idea of which high-level features would be available by the Social Media Expo. Let's imagine that his roadmap indicates that Release 1.0 will focus on "basic learning" and "basic filtering" features corresponding to the following backlog items:

> As a typical user I want to teach SR4U what types of reviews to discard so that SR4U will know what characteristics to use when discarding reviews on my behalf.
>
> As a typical user I want a simple, Google-like interface for requesting a review search so that I don't have to spend much time describing what I want.

At release planning, these items will be too large to work with. To refine them Roger and his team would conduct a user-story-writing workshop (see Chapter 5) as part of the release-planning meeting or perhaps a separate story-writing workshop before the release-planning meeting. The results of this workshop would be many more detailed product backlog items, such as these:

> As a typical user I want to tell SR4U to ignore reviews that contain specific keywords that I feel show bias in a review so that I don't see any reviews containing those keywords.
>
> As a typical user I want to select a category of product or service so that I can help SR4U focus only on relevant reviews.

Once the stories are small enough, the team would estimate them (see Chapter 7) to communicate a rough idea of the cost. (Some amount of estimation is necessary for initial release planning. And as new stories emerge during the release, they too will need to be estimated for ongoing release-planning activities.) The release-planning participants would then prioritize the estimated stories based on the release goal and constraints. As the product backlog is reprioritized, the participants should be vigilant to ensure that the minimum releasable feature set is always identified and agreed upon.

Refine Minimum Releasable Features (MRFs)

As I described in Chapter 17, the minimum releasable features represent the smallest set of "must-have" features, the ones that simply have to be in the release if we are to meet customer value and quality expectations. An important part of release planning is to diligently reevaluate and refine what are truly the MRFs for the release. As we obtain fast feedback from our sprints and acquire validated learning, we are constantly adjusting the MRFs.

A problem I frequently see in organizations is an inability to agree on what constitutes the MRFs. Multiple competing stakeholders simply might not agree. Having poorly defined MRFs or MRFs that people only passive-aggressively agree with interferes with clear decision making during release planning. For example, we're running out of time; which feature should we drop? Lack of clarity regarding the MRFs might complicate this decision.

In Scrum, the product owner is ultimately responsible for defining the MRFs. Of course, he can and will do this in close collaboration with the proper stakeholders and the Scrum team.

For some, the MRFs concept may feel counterintuitive—why not try to deliver the largest set of features in a release instead of the smallest? The simple answer is that the largest set of features probably costs the most money, takes the most time, and has the most risk. Conversely, the smallest possible feature set should cost the least money, take the least time, and have the least risk. Thinking minimally better aligns us with the principle of delivering smaller and more frequent releases, as described in Chapter 14.

The MRFs should be defined with knowledge of the feature sizes, as determined during product backlog grooming. Not everyone agrees. Some believe that the MRFs should be defined independently of cost—meaning the MRFs are the minimum releasable features that will meet the users' value threshold for this release (and this determination is made independently of cost data). Initial MRFs can be envisioned without cost data, but because all of our release-planning decisions need to be made within a sensible economic framework, knowing feature costs provides a critical check on the economic viability of the MRFs. If we determine that the MRFs are not economically viable, perhaps it is time to pivot.

Sprint Mapping (PBI Slotting)

In each sprint the team works on a set of product backlog items. The team and product owner don't decide which specific product backlog items to work on in a given sprint until sprint planning. Does that mean we should give no consideration to mapping product backlog items to sprints before sprint-planning meetings?

Absolutely not! Some teams believe that a quick, early mapping (or slotting) of near-term product backlog items into sprints is helpful. For example, mapping out a

few sprints in a multiteam environment might help the teams better coordinate their work.

To do this mapping we need an appropriately detailed, estimated, and prioritized product backlog. Using our team's velocity, we can approximate a set of product backlog items for each sprint by grouping together items whose aggregate size roughly equals the team's average velocity. The result might look like the left side of Figure 18.5. Some people prefer to show the sprint map horizontally (right side of Figure 18.5) to more closely resemble a timeline. I have seen some teams place the horizontally oriented sprint map of a Scrum team above a standard project plan (a Gantt chart, for example) that describes the work of non-Scrum teams to better visualize the alignment and touch points between the Scrum development work and the non-Scrum work.

When developing with one Scrum team we might do this mapping during initial release planning to get a rough idea of when certain features within the release will be created. This mapping activity might also cause us to reorganize the product backlog to group items in ways that are more natural or efficient. We also might choose to

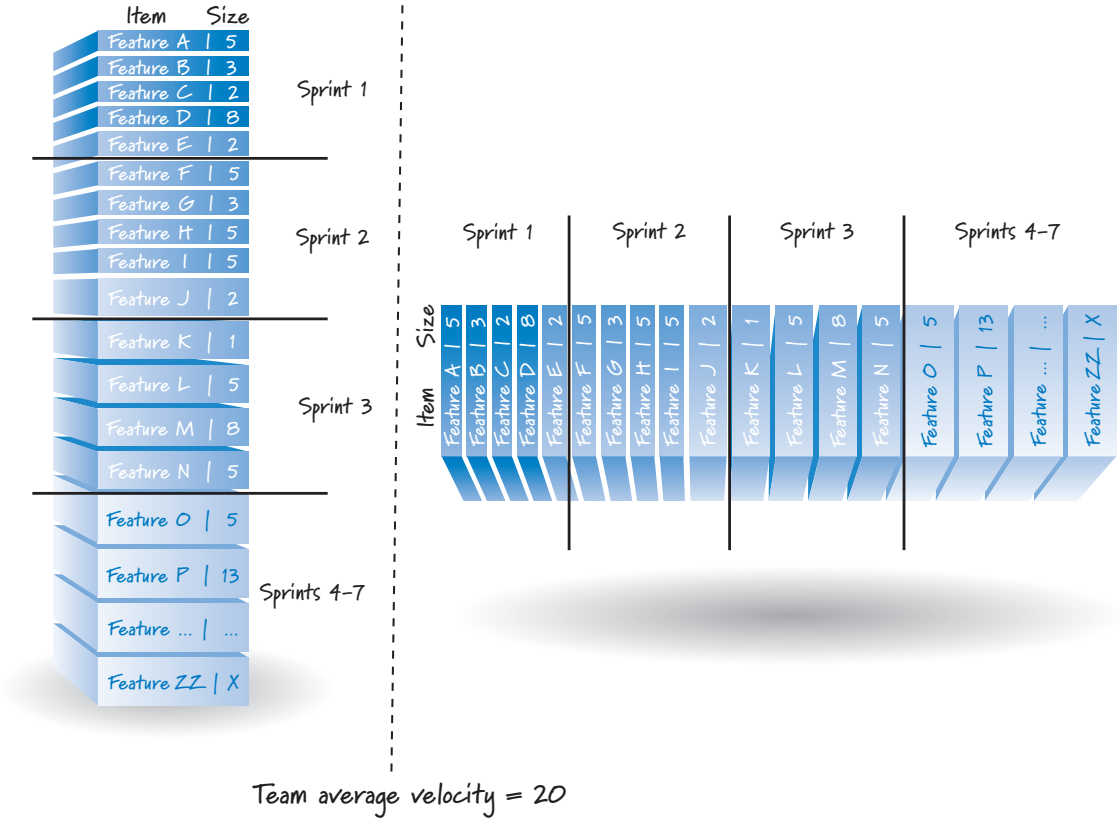

FIGURE 18.5 Mapping product backlog items to sprints

reorganize the work to ensure that the results at the end of a sprint are sufficient for us to get validated learning and actionable feedback.

When developing with multiple teams, we may want to do some forward-looking mapping of items to sprints to help manage inter-team dependencies. One approach is to use a form of rolling look-ahead planning (Cohn 2006), where each team considers the needed backlog items not only for the upcoming sprint but for at least two (and sometimes more) future sprints. This way, when more than one Scrum team is involved, the teams can know what team is going to work on which items and roughly when.

For example, assume that SR4U will have three Scrum teams. Team 1 focuses on the end-to-end processing of user requests. This team enables users to specify a review query, run the query, and get review results. Team 2 focuses on the AI engine, which has the logic for how to analyze and discriminate among reviews. Team 3 focuses on connecting to different Internet data sources for retrieving candidate reviews.

These three teams must coordinate their efforts to make sure that the minimum releasable features are produced and available in time for the Social Media Expo. It makes sense for all three teams to participate in joint release planning.

During an initial release-planning meeting each of the teams provides an idea of when it will work on its product backlog items. In the ensuing discussion, team 1 might say, "We think we'll be ready to create the *ignore reviews with specific keywords* feature in sprint 2. However, we will need team 3 to be able to retrieve data from at least one Internet source either before that sprint begins or very early during sprint 2." Members of team 3 can then examine their sprint map to see if they are currently planning to have at least one source available by sprint 2. If not, the two teams can discuss the dependency and see what modifications one or both of the teams need to make.

If we choose to do some early mapping, we must realize that this mapping can and will evolve during the creation of the release. Ultimately the decision as to which features each team will work on in a given sprint is made at the last responsible moment—the sprint planning for that sprint.

Alternatively, a preference of many organizations (especially those doing product development using only one team) is to perform little or no early mapping of product backlog items to sprints. Teams in such companies believe that the effort to produce the mapping is not justified by the value that it delivers. For these teams, the initial release-planning meeting would not involve the mapping step, at least not in any significant way.

Fixed-Date Release Planning

As I mentioned earlier, many organizations that use Scrum prefer to use fixed-date releases. Table 18.2 summarizes the steps for performing fixed-date release planning.

Let's use SR4U as an example. Release 1.0 is tied to the start of the Social Media Expo, which starts on Monday, September 26. The company decides that having an

TABLE 18.2 Steps for Performing Fixed-Date Release Planning

Step	Description	Comments
1	Determine how many sprints are in this release.	If all sprint lengths are equal, this is simple calendar math because you know when the first sprint will start and you know the delivery date.
2	Groom the product backlog to a sufficient depth by creating, estimating the size of, and prioritizing product backlog items.	Because we are trying to determine which PBIs we can get by a fixed date, we need enough of them to plan out to that date.
3	Measure or estimate the team's velocity as a range.	Determine an average faster and an average slower velocity for the team (see Chapter 7).
4	Multiply the slower velocity by the number of sprints. Count down that number of points into the product backlog and draw a line.	This is the "will-have" line.
5	Multiply the faster velocity by the number of sprints. Count down that number of points into the product backlog and draw a second line.	This is the "might-have" line.

initial version to demonstrate at this conference is an excellent first milestone on the path to realizing the product vision.

Although we just imagined SR4U with three teams, for this example, let's go back to the initial assumption that just one team would develop the product. Because Roger and the team want to begin sprint 1 the first week in July and finish by September 23 (the Friday before the Expo starts), they can easily calculate how many sprints they will need to perform in this release. Assuming that the length of each sprint is the same throughout this release, which is the normal case with Scrum, SR4U Release 1.0 has six, two-week (ten-day) sprints. Figure 18.6 maps these sprints onto the calendar.

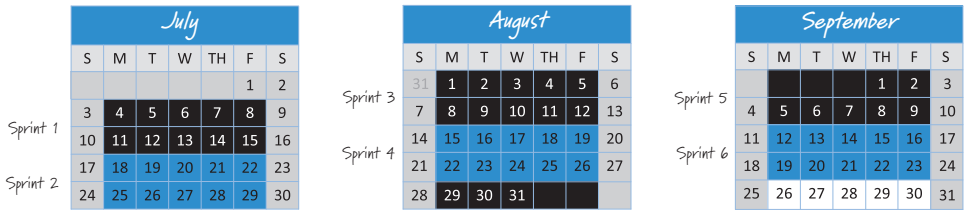

FIGURE 18.6 Sprint calendar for SR4U Release 1.0

Next they determine how much work the team can get done in six sprints. Using the approach I discussed in Chapter 7, let's say they calculate the velocity of the team to be between 18 and 22 story points per sprint. Therefore, Roger and the team should be able to complete between 108 and 132 story points of work during this release.

Now they need to determine what features this range of story points represents. At the end of product planning, Roger and the team had a high-level product backlog with some epic- and theme-level user stories. As I discussed earlier, the SR4U team then conducted a user-story-writing workshop to create more detailed product backlog items. The team then estimated them, and the product owner, with input from the development team and stakeholders, prioritized them.

As part of this process, Roger and the team had to determine the set of must-have features that make up the MRFs. As a rule of thumb, on a fixed-date release I like the MRFs to require less than 100% of the time allocated to the release. I prefer somewhere closer to 60% to 70% for at least two reasons:

- If we run out of time for the release and all of the features targeted for the release are must-have items, which feature should we drop? By definition, if the features are all must-haves, we wouldn't be able to drop any. If we define the release to have about 60% to 70% must-have features, with the remaining scope being nice-to-have features, we can drop nice-to-have features if we have to drop scope.
- If we allocate 100% must-have features to the release, what happens during the release when an emergent must-have feature appears? In other words, a feature we didn't know about earlier presents itself and it absolutely must be included to have a viable release. How would we accommodate it? If we defined the release to include some nice-to-have features, we could drop one or more of them to include the new, emergent must-have feature.

The end result is a product backlog that structurally looks like Figure 18.7. This figure shows the total product backlog as Roger and the team understand it today, including themes and epics that are not planned for this release.

Roger and the team can then apply the results from the earlier velocity calculation, where the team estimated it would be able to complete between 108 and 132 points' worth of work over six sprints. The team can visualize where in the backlog it can get to by counting down a total of 108 points from the top and then counting down to a total of 132 points (see Figure 18.8).

You'll notice that these two lines split the product backlog into three sections (will have, might have, won't have). This approach illustrates how we can give a range answer to the question "What will I get by the release date?" Early in the release it is difficult to give a very precise answer to this question. The range answer is accurate and also communicates the uncertainty we have in the answer—the broader the range, the less certain we are.

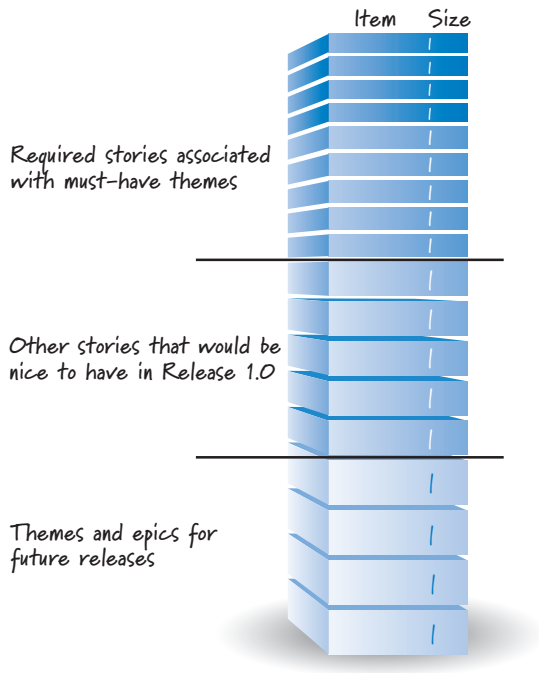

FIGURE 18.7 Product backlog ready for release planning

To understand whether they are in good shape with this release plan, Roger and the team need only to overlay the must-have line (from Figure 18.7) onto the product backlog (from Figure 18.8). Some possible results are shown in Figure 18.9. Notice that the must-have line separates the minimum releasable features that are above the line from the rest of the product backlog.

The left-most product backlog of Figure 18.9 communicates a very positive situation. You can interpret it as "We will have our must-have features." We should proceed with the release.

The middle backlog of Figure 18.9 can be interpreted as "We will have most of our must-have features, but we might or might not have all of them." There is clearly more risk associated with this scenario than the previous scenario. One option is to accept the risk that we won't get all of the must-have items and move on. Because we are planning to learn fast, we might decide to start this release and complete a few sprints. At that point, we could reevaluate where we are and then make a decision to continue with the release or kill it (as I discussed in previous chapters). Also, feedback from the work already completed might indicate that some of the features originally included in the MRFs aren't really must-have items after all and we are actually in good shape.

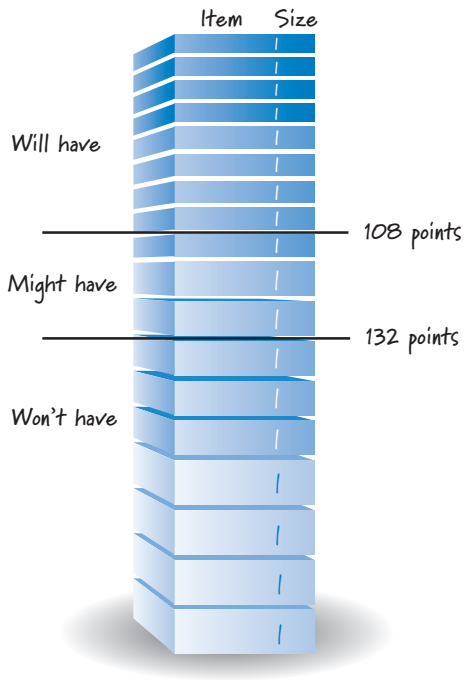

FIGURE 18.8 Determining the range of features on a fixed-date release

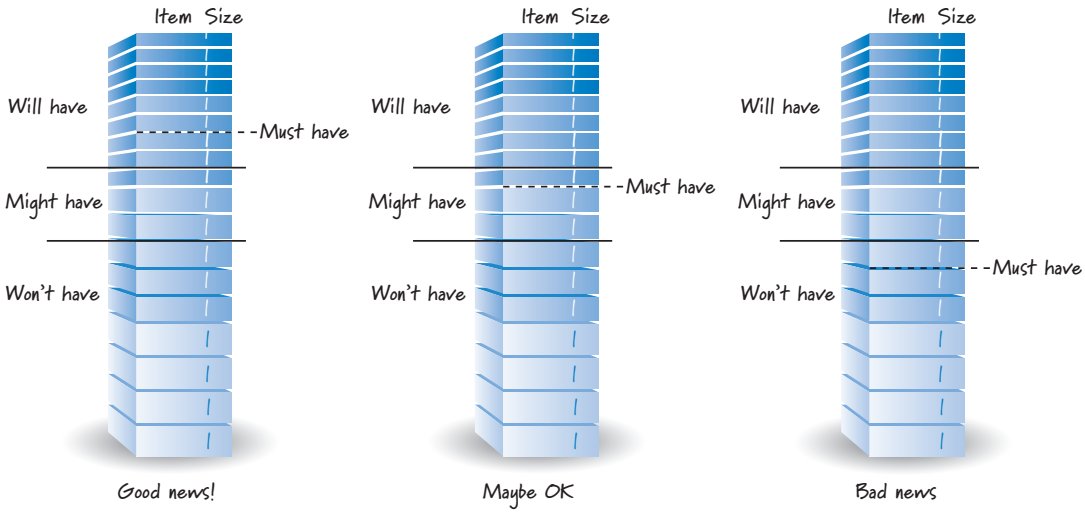

FIGURE 18.9 Location of must-have features relative to the range of deliverable features

Alternatively, we can consider setting a new, later release date that we could easily calculate, or we can suggest adding more people to the development effort to increase velocity (if we think that might help). At this point, some organizations might choose to accrue technical debt by having the team cut corners to ensure that all of the must-have features are delivered by the due date, albeit at a reduced quality level. However, if technical debt is taken on in the current release, it should be paid off in the next release, which will reduce the amount of value delivered then.

The right-most backlog in Figure 18.9 can be interpreted as "We won't have our must-have features." Perhaps we shouldn't proceed with this release, or maybe we should change the release date or consider adding more resources. If we choose to accrue technical debt in this scenario, it will probably be a lot of debt.

Of course, assuming that we proceed with the release, we must revisit our release plan every sprint to update it based on our current knowledge.

For example, at the end of each sprint we have an additional velocity data point, which for a new team without much historical velocity data, or a team doing radically different work from what it is used to, will cause us to recalculate the average points per sprint the team can get done. And, as you might expect, the items in the product backlog could change. New items could emerge and other items could be moved out of this release or deleted altogether as we learn that we don't need them now or ever. To visually communicate the revised release plan, we would redraw a picture similar to Figure 18.8.

Fixed-Scope Release Planning

Although fixed-date releases are very common with Scrum, what if for your product the scope truly is more important than the date? What if you have a large set of must-have features in your minimum releasable features set and you're willing to slip the delivery date to get them all?

If you're in this situation, have you truly winnowed down the must-have features to be the absolute bare minimum? I occasionally hear things like "But we're implementing a standard and you can't ship an implementation of half a standard." While this is perhaps true, in most standards there are still likely to be optional parts that we don't have to implement now (for example, think about web browser support for changing or emerging HTML or CSS standards). In other instances we might be able to go to market with less than the full implementation of the standard and let customers know which parts we support and which parts we don't.

My point is that if we think incrementally and aggressively target the true minimum releasable features, we can usually turn a fixed-scope release into a set of smaller fixed-date releases. When the set of minimum releasable features becomes small, another constraint (like time) typically becomes the dominant constraint.

Let's say that we have winnowed down the must-have features to the bare minimum and our principal release constraint is still focused on when we can we get those features. In this case we perform release planning as outlined in Table 18.3.

If we're doing a fixed-scope release, we must know what the features are at the start of the release. We might know these features if we are building a simple or familiar product. When developing innovative products, however, many features will emerge and evolve during the development effort. We certainly have some idea of the desired features up front, so we'll use them in our initial release planning. However, we must be prepared to continuously revise our release plan as our understanding of the required features changes.

If we perform a release-planning meeting at the start of the release, we must first groom the product backlog as we did during fixed-date planning. A difference is that during fixed-date planning we try to have fewer than 100% must-have items in the release to buffer against uncertainty. In fixed-scope planning we want the entire scope for the release to be must-have features. Our goal on fixed-scope releases is to get all of the must-have features completed in a timely way. If an emergent must-have feature appears, we will simply add it to the scope of the release and push out the release date.

During fixed-date planning, we know precisely how many sprints we will have. During fixed-scope planning, we need to calculate the number of sprints required to deliver the fixed set of features.

TABLE 18.3 Steps for Performing Fixed-Scope Release Planning

Step	Description	Comments
1	Groom the product backlog to include at least the PBIs we would like in this release by creating, estimating the size of, and prioritizing PBIs.	Because this is a fixed-scope release, we need to know which PBIs are in the fixed scope.
2	Determine the total size of the PBIs to be delivered in the release.	If we have a product backlog of estimated items, we simply sum the size estimates of all of the items we want in the release.
3	Measure or estimate the team's velocity as a range.	Determine an average faster and an average slower velocity for the team.
4	Divide the total size of the PBIs by the faster velocity and round up the answer to the next integer.	This will tell us the lowest number of sprints required to deliver the features.
5	Divide the total size of the PBIs by the slower velocity and round up the answer to the next integer.	This will tell us the highest number of sprints required to deliver the features.

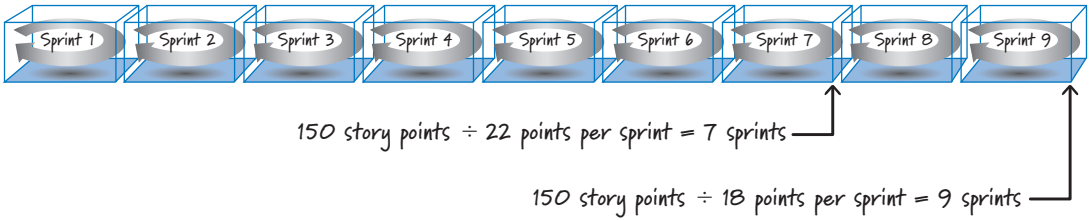

FIGURE 18.10 Results of fixed-scope planning

To perform the math we need the velocity range for our team (as we did with fixed-date planning). Let's say our team's velocity on two-week sprints ranges between 18 and 22 story points. To answer the question of when we will get the fixed set of features, we sum the sizes of all of those features and then divide by our team's higher and lower velocities. The result is a range of sprints within which delivery will take place.

Let's say we want 150 story points of features in the next release. If we divide 150 by 18 (our team's slower velocity) and round up, we get nine sprints. If we divide 150 by 22 (our team's faster velocity) and round up, we get seven sprints. We can visualize this as shown in Figure 18.10.

Notice that once again we give a range answer to the question we are being asked. In this case the question is "How many sprints will you need to complete a release with 150 points of work?" Our answer will be seven to nine sprints. Because these are two-week sprints, our answer also could be stated as 14 to 18 weeks.

Calculating Cost

Calculating costs on either a fixed-date or fixed-scope release is easy (see Table 18.4).

TABLE 18.4 Calculating the Cost of a Release

Step	Description	Comments
1	Determine who is on the team.	Assume that the team composition doesn't materially change either during a sprint or from sprint to sprint.
2	Determine the sprint length.	Assume that all sprints have the same length.
3	Based on team composition and sprint length, determine the personnel costs of running a sprint.	This is simple if previous assumptions are true, and only slightly more complicated if team composition or sprint length fluctuates.

continues

TABLE 18.4 Calculating the Cost of a Release (*Continued*)

Step	Description	Comments
4a	For a fixed-date release, multiply the number of sprints in the release by the cost per sprint.	The result is the fixed personnel cost for this release.
4b	For a fixed-scope release, multiply both the high and low number of sprints by the cost per sprint.	The results are the range of personnel costs associated with the release. One represents the lower amount the release should cost and the other the higher amount it should cost.

Let's assume that the composition of the team assigned to the development effort is reasonably stable. In other words, we aren't taking people off of the team or adding new people to the team. And, if we do, the changes are small and the people we move around get paid something reasonably similar.

Based on these assumptions, we can easily determine the cost per sprint, because we know who is on the team and the length of the sprints. If we remove other costs (like capital costs) from this discussion, which often is reasonable because the majority of software development costs are the cost of the people, personnel cost is a pretty good surrogate for the overall cost per sprint.

To finalize the calculation, we need to know the number of sprints within the release. On a fixed-date release we know exactly the number of sprints, so we multiply the number of sprints by the cost per sprint to determine the release cost.

On a fixed-scope release, we have a range of sprints. In our previous example we calculated a range of seven to nine sprints, so the cost for this release will range from seven to nine times the cost per sprint. Most organizations will budget at the high end of this range because the release may in fact take nine sprints to complete. If we budgeted for only seven sprints, we might have insufficient funds to complete the release.

Another approach to calculating cost can be used if you know your historic cost per story point. If you have data that indicates how many points of work were completed during a previous period of time (say, a year) and you divide that into the loaded labor cost of the team, you will know your cost per point. If it is reasonable to assume that the same cost per point would apply to the current release, you can roughly estimate the cost for a 150-point release (by multiplying 150 by the historic cost per point) even before doing initial release planning.

Communicating

An important aspect of release planning is communicating progress. Although any highly visible way of communicating progress can be used, most teams use some

form of burndown and/or burnup chart as their principal information radiator of release status. Let's look at how to communicate release status on both fixed-scope and fixed-date releases.

Communicating Progress on a Fixed-Scope Release

On a fixed-scope release we have an idea of the total scope of work we wish to achieve. The goal is to communicate how we are progressing toward completing that work.

Fixed-Scope-Release Burndown Chart

A **burndown chart** for a fixed-scope release shows the total amount of unfinished work that remains each sprint to achieve the current release goal. In this type of chart, the vertical axis numbers are in the same units we use to size our product backlog items (typically story points or ideal days). The horizontal axis represents sprints (see Figure 18.11).

Using the example from earlier in the chapter, we have 150 story points at the start of development (at the end of initial release planning), which is the same as the start of sprint 1. At the end of each sprint we update this chart to show the total amount of work remaining within the release. The difference between the amount of work remaining at the beginning of a sprint and the work remaining at the end of the sprint represents the sprint velocity. This is plotted as the "Actual" line in Figure 18.11.

We can also show projected outcomes on the burndown chart. In Figure 18.11 there are three lines predicting when the release might be done, each corresponding to a predicted team velocity. If the team is able to work at its higher velocity of 22 points per sprint, the team would finish by the end of seven sprints. If the team

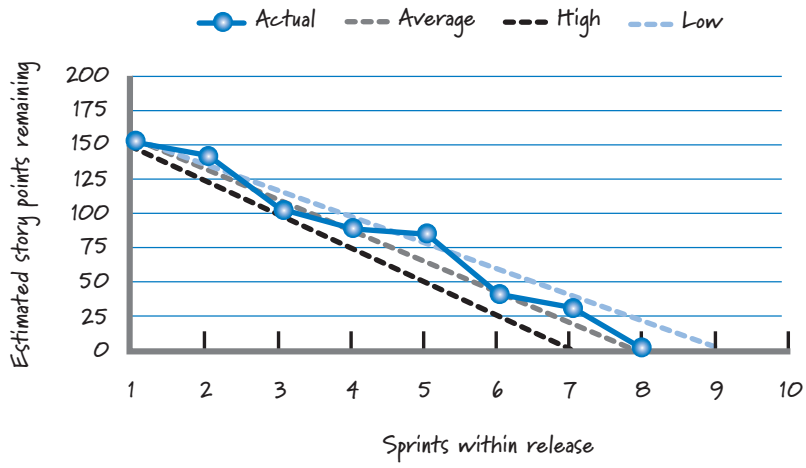

FIGURE 18.11 Fixed-scope-release burndown chart

operates at its lower velocity of 18, it might need a total of nine sprints. And, if the team operates at its average velocity of 20, it would need eight sprints.

There are several variations on the basic burndown chart; however, they are all similar in that they show each sprint the cumulative size of the work remaining to achieve the release goal.

Fixed-Scope-Release Burnup Chart

A **burnup chart** for a fixed-scope release shows the total amount of work in a release as a goal or target line and our progress each sprint toward achieving that goal (see Figure 18.12). The horizontal and vertical dimensions of the chart are identical to those of the release burndown chart.

In this chart, at the end of every sprint we increment the cumulative story points completed by the total story points completed in that sprint. The goal is to burn up to achieve the target number of story points in the release. And, like the release burndown chart, this chart shows the same three predictive lines indicating the likely number of sprints to achieve the target.

Some people prefer to use the burnup format because it can easily show a change in scope for the release. For example, if we add more scope in the current release (so the release really isn't fixed scope!), in the sprint where the new scope is added we simply move the target line up to indicate that a new, higher target exists from this point forward (see Figure 18.13).

It is also possible to show a change in scope on a release burndown chart (see Cohn 2006).

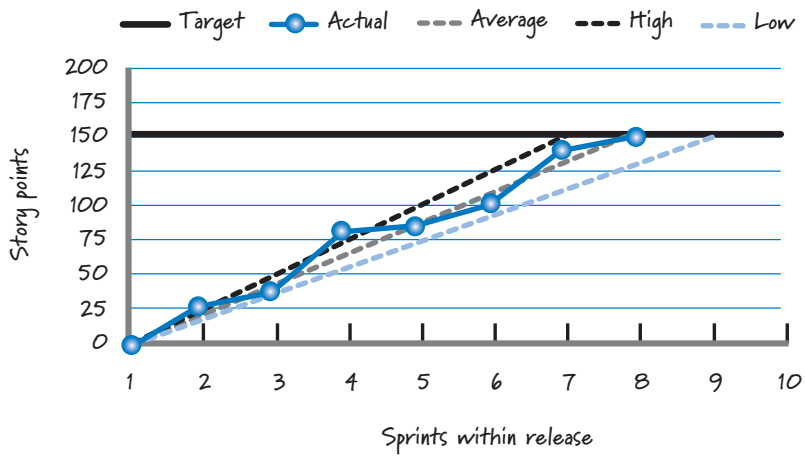

FIGURE 18.12 Fixed-scope-release burnup chart

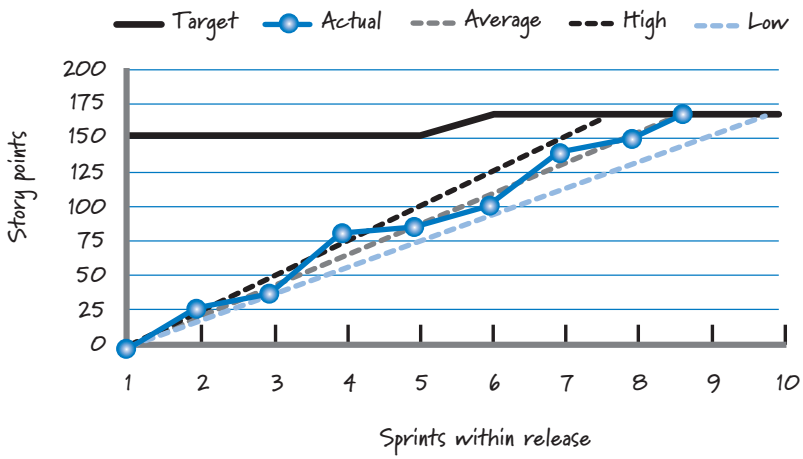

FIGURE 18.13 Variable-scope-release burnup chart

Communicating Progress on a Fixed-Date Release

With a fixed-date release we know the number of sprints in the release, so our goal is to communicate the range of features we expect to complete and our sprint-by-sprint progress toward the range. The traditional burndown and burnup charts aren't effective tools for fixed-date planning because they assume you know how much total scope you have to burn down or burn up. Remember this is fixed-date planning, so we are trying to calculate and communicate over time the narrowing range of scope that can be delivered by a fixed date.

Figure 18.9 shows how to visualize a range of features we expect to achieve in a fixed-date release. If we update the charts in Figure 18.9 at the end of each sprint, we have a very effective way of communicating the projected range of features we will complete by the fixed release date. We will also have an understanding of how likely we are to get the must-have features by the release date.

If we want to maintain one chart that shows our historical progress toward achieving the final scope, we can create a specialized burnup chart in the form of Figure 18.14.

This chart has all of the same elements as the charts in Figure 18.9. However, in Figure 18.14, the product backlog is inverted (intentionally positioned upside down) so that instead of the highest-priority item being physically positioned at the top, it is now at the bottom of the product backlog. Lower-priority items are now found higher up in the backlog. Inverting the backlog in this fashion eliminates the problem of having to know the scope of the product backlog items in the release (which traditional release burndown and burnup charts require).

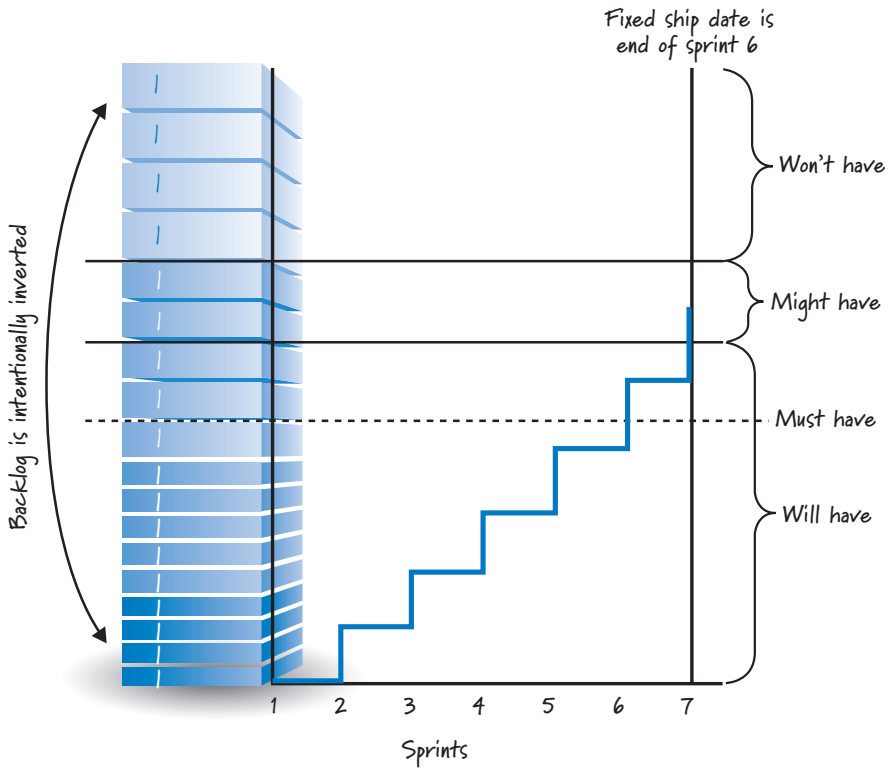

FIGURE 18.14 Fixed-date-release burnup chart (with inverted product backlog)

The chart shows the projected range of features that we expect to have by the end of sprint 6 (beginning of sprint 7). Each sprint we burn up the chart to show the features completed in that sprint. So at the end of sprint 1 (beginning of sprint 2) there is a vertical line whose length indicates how many features we completed in sprint 1. Using this approach allows us to see how we are progressing toward hitting our target range of features, as well as how we are progressing toward completing the must-have features. For simplicity, there are no trend lines on this graph, but they could easily be added to extrapolate from earlier sprints what scope we are likely to end up with.

Closing

In this chapter I expanded the description of release planning by discussing when release planning takes place, who is involved, what activities take place, and the elements of the resulting release plan. I also discussed the details of how to do both

fixed-date and fixed-scope release planning and how to communicate progress during a release.

This chapter concludes Part III. In the next chapter I will discuss the next level of planning: sprint planning, which I include with Part IV to group together all of the chapters related to sprint-specific activities.